

Kubernetes A Visual Guide

**Sample Book
Chapters 1 to 2**

Complete Foundational Guide

Introduction

This book is designed for readers who are taking their first steps with Kubernetes, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

Read the chapters in order on your first pass. The sequence builds the required background step by step: the Kubernetes mental model comes first, followed by application operation, reliability, security, troubleshooting, reusable delivery, extension mechanisms, and cluster lifecycle responsibilities.

- Review **What You'll Be Able to Do** at the beginning of each chapter before reading the chapter itself. These outcomes identify the concepts and practical skills that deserve the most attention.
- Use the diagrams to understand relationships, processes, boundaries, and structures that are difficult to hold in mind from prose alone. Read the surrounding text for assumptions, limitations, and operational meaning.
- Try the code and hands-on examples whenever a suitable learning cluster is available. Check the stated prerequisites, predict the result before running a command, and compare the actual result with the expected result.
- Use each **Chapter Summary** and set of **Review Questions** to confirm that you can explain the fundamentals in your own words rather than merely recognize commands or YAML fields.
- Treat advanced-topic overviews as signposts. You do not need to memorize every tool, extension, or administrative detail at this stage; remember what problem it solves and where to return when the topic becomes relevant.

This is a beginner-friendly foundation, not a substitute for environment-specific procedures. When applying an example to real systems, confirm the current cluster version, installed controllers, organizational policy, and official documentation for the technologies involved.

Table of Contents

[Introduction](#)

[How to Use This Book](#)

[Part I — Build the Correct Mental Model](#)

[Chapter 1 — Kubernetes in One Picture](#)

- [The big picture](#)
- [The operational problems Kubernetes addresses](#)
- [Desired state, actual state, and reconciliation](#)
- [A conceptual request path](#)
- [What “self-healing” means](#)
- [What Kubernetes does not provide by itself](#)
- [Shared responsibility](#)
- [The first practical mental model](#)

[Chapter 2 — Container, Linux, and Network Foundations](#)

- [Kubernetes orchestrates processes](#)
- [Image, registry, container, and process](#)
- [A container is not a small virtual machine](#)
- [PID 1, signals, and exit codes](#)
- [Command and arguments](#)
- [CPU and memory are different kinds of resources](#)
- [Standard output and standard error](#)
- [Networking vocabulary](#)
- [A local container example](#)
- [Ephemeral and persistent data](#)
- [Environment variables](#)
- [YAML essentials](#)
- [Foundation check](#)

Main Text

Kubernetes is an API-driven control system for running containerized workloads. You describe the state you want—such as three instances of an application, a stable network endpoint, and a mounted configuration file—and Kubernetes continuously works to make the cluster match that description.

This mental model is more important than memorizing commands. A manifest is not a script that performs a fixed sequence of steps. It is a declaration that several independent control loops may act on over time. A successful command therefore means that the Kubernetes API accepted a request; it does not necessarily mean that the application is already running or healthy.

The examples use stable Kubernetes APIs and a generic namespace named `learning`. They assume access to a learning cluster and a compatible `kubectl` command-line client. Features that depend on a cloud provider, storage system, ingress controller, or Gateway API implementation are explicitly identified as environment-dependent.

Part I — Build the Correct Mental Model

Chapter 1 — Kubernetes in One Picture

What You'll Be Able to Do

After this chapter, you will be able to:

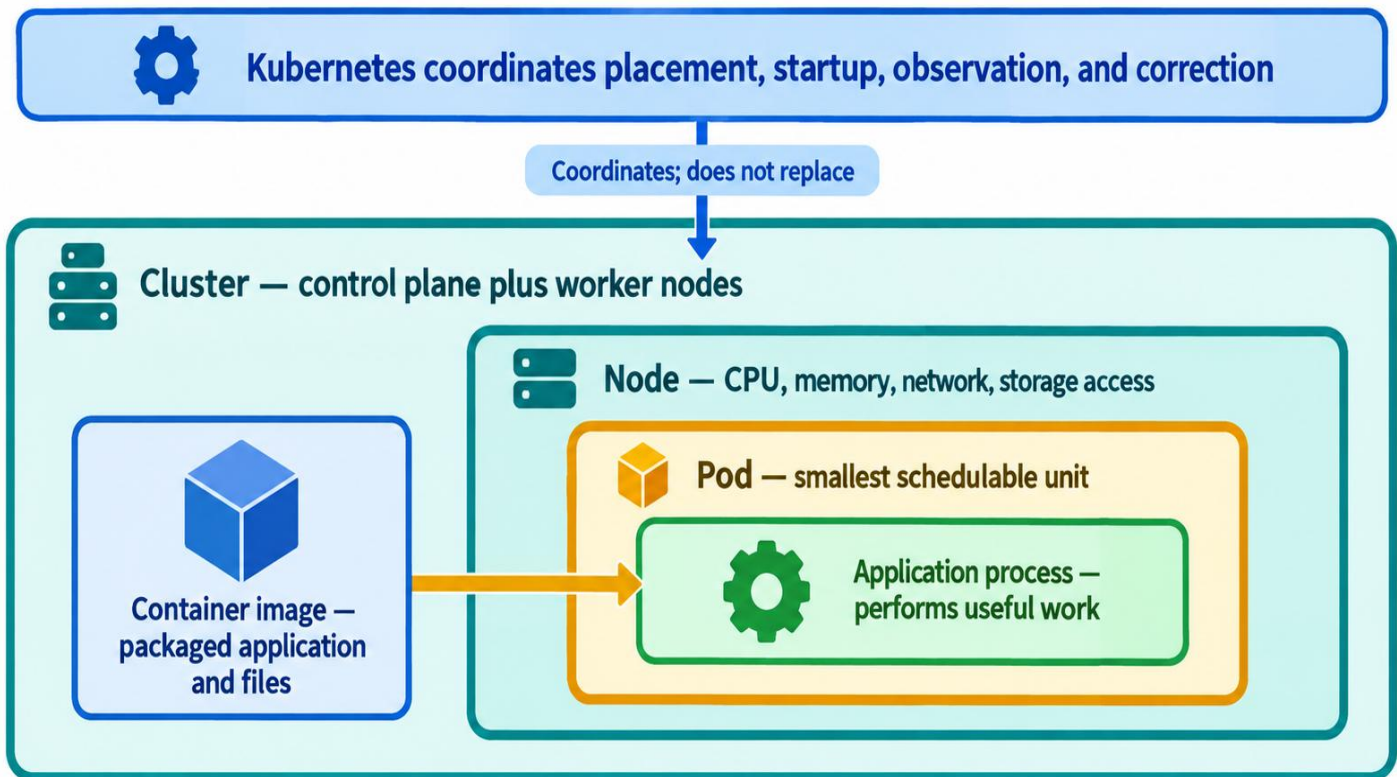
- Explain the relationship among an application, container image, Pod, node, and cluster.
- Describe desired state, actual state, reconciliation, and the practical meaning of self-healing.
- Trace a workload request conceptually and identify where Kubernetes responsibility ends.

The big picture

Kubernetes coordinates applications by comparing desired state with actual state and correcting the difference. It does this through API objects, controllers, a scheduler, and node-level agents.

Chapter 1: Kubernetes in One Picture

1-1 Kubernetes in one layered picture



001

Figure 1-1: Kubernetes in one layered picture

A useful first picture contains five layers:

1. An **application** performs useful work, such as serving an HTTP request.
2. A **container image** packages the application and the files it needs to start.
3. A **Pod** is Kubernetes' smallest schedulable execution unit and holds one or more closely related containers.
4. A **node** is a machine that provides CPU, memory, networking, and storage access for Pods.
5. A **cluster** is the control plane plus the nodes managed as one system.

Kubernetes does not replace the application or the image. It decides where Pods should run, starts their containers through a container runtime, observes the results, and reacts when the observed state differs from the declared state.

The operational problems Kubernetes addresses

Running one container manually is straightforward. Operating many applications across many machines creates recurring problems:

- **Placement:** Which machine has enough capacity and satisfies the workload's constraints?
- **Restart and replacement:** What should happen when a process exits or a machine disappears?
- **Replication:** How can several interchangeable instances be kept running?
- **Rollout:** How can a new application version replace an old one in a controlled way?
- **Service discovery:** How can clients find changing application instances?
- **Configuration:** How can one image be used with different settings in different environments?
- **Storage attachment:** How can a workload request durable storage without knowing the infrastructure details?
- **Scaling:** How can the number of instances be changed without manually starting each one?

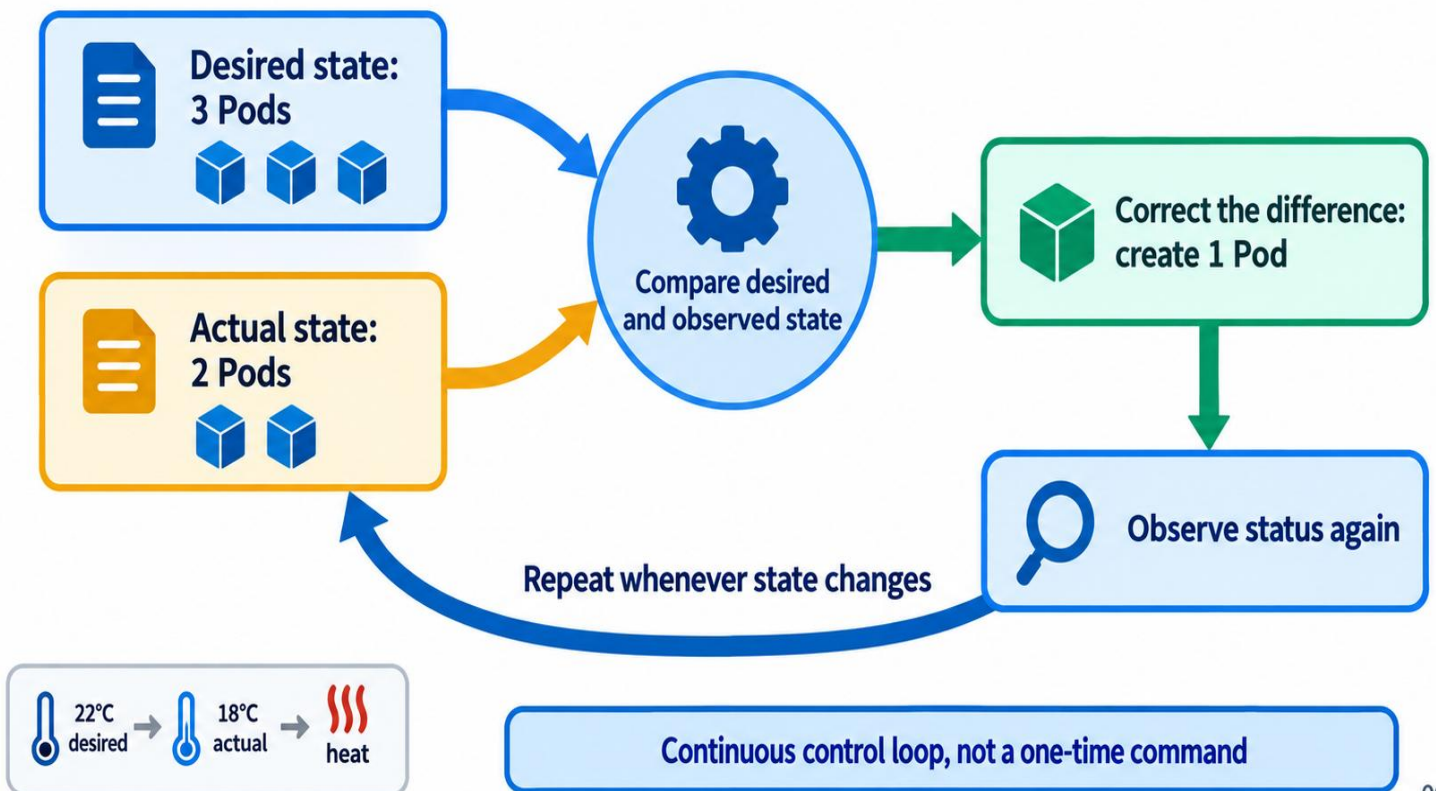
Kubernetes supplies common abstractions for these problems. It does not remove the need to make application-specific decisions. For example, Kubernetes can restart a failed process, but it cannot decide whether an incorrect business calculation is safe.

Desired state, actual state, and reconciliation

Desired state is the state declared in Kubernetes objects. **Actual state** is what currently exists in the cluster. **Reconciliation** is the repeated process of moving actual state toward desired state.

Chapter 1: Kubernetes in One Picture

1-3 Desired state, actual state, and reconciliation



002

Figure 1-2: Desired state, actual state, and reconciliation

A thermostat is a useful analogy:

- You set 22°C. That is the desired state.
- The room is currently 18°C. That is the actual state.
- The thermostat detects the difference and turns on the heater.
- It keeps checking rather than assuming that one action permanently solved the problem.

A Kubernetes Deployment works similarly. Suppose its specification says that three application Pods should exist:

1. The Deployment controller observes that only two matching Pods exist.
2. It requests another Pod through the Kubernetes API.
3. The scheduler selects a node for the new Pod.
4. The node's kubelet starts the required container.
5. Status information flows back through the API.
6. Controllers continue observing because a Pod or node may fail later.

This is why **Kubernetes is better understood as a collection of control loops** than as a command executor.

A conceptual request path

Consider a systems engineer applying a manifest for an HTTP application:

Chapter 1: Kubernetes in One Picture

1-4 From manifest to two running Pods

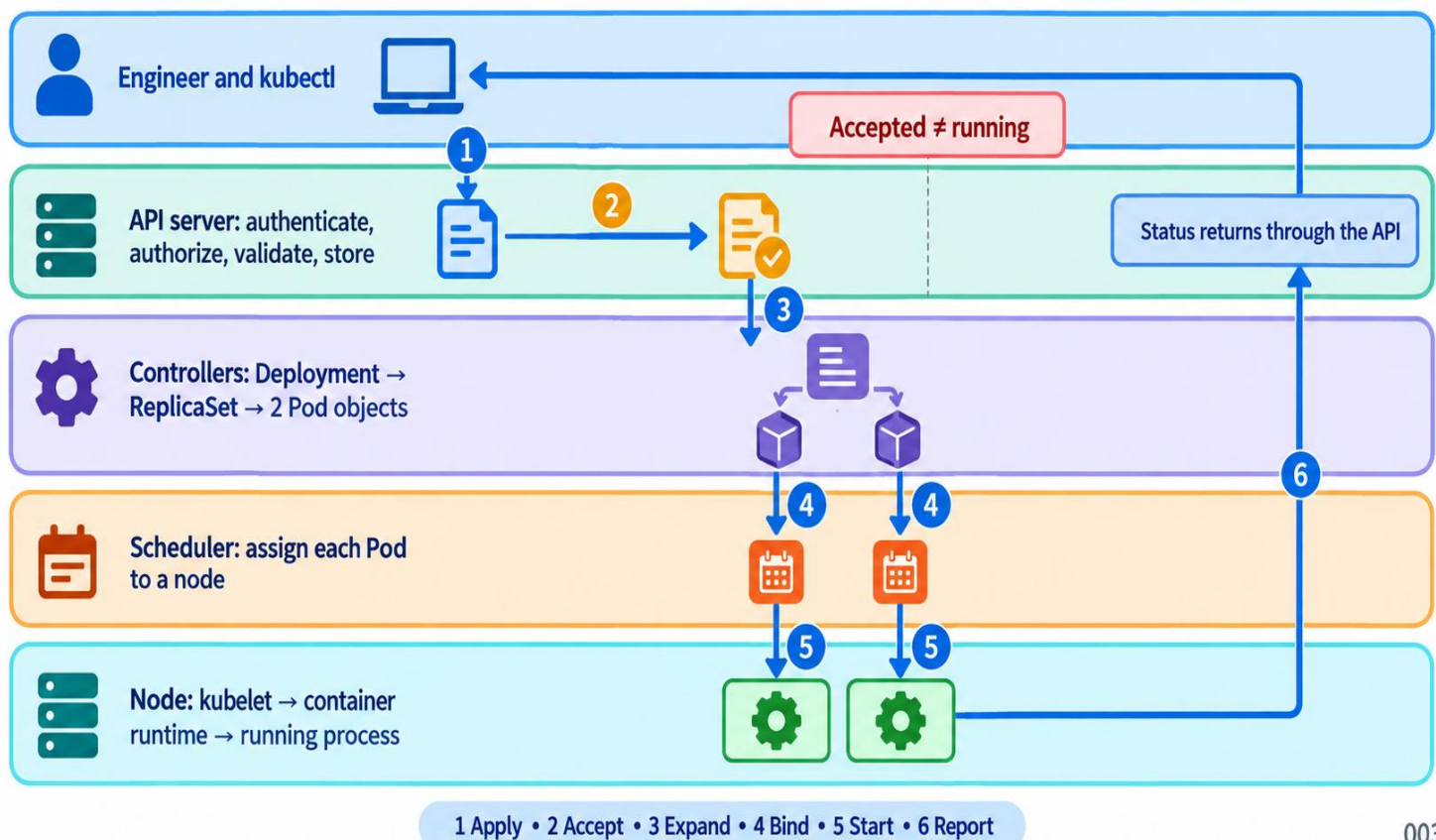


Figure 1-3: From manifest to two running Pods

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-demo
spec:
  replicas: 2
```

```
selector:
  matchLabels:
    app: web-demo
template:
  metadata:
    labels:
      app: web-demo
  spec:
    containers:
      - name: web
        image: nginx:1.27.4-alpine
```

The important lines are:

- `kind: Deployment` asks for a controller-managed stateless workload.
- `replicas: 2` declares two interchangeable Pod instances.
- `selector` and the Pod template labels connect the Deployment to its Pods.
- `image` identifies the packaged application to run.

At a conceptual level, the request proceeds as follows:

1. `kubectl` sends the manifest to the API server.
2. The API server authenticates and authorizes the request, validates the object, and stores the accepted state.
3. The Deployment controller notices the new desired state and creates a ReplicaSet.
4. The ReplicaSet controller creates two Pod objects.
5. The scheduler assigns each unscheduled Pod to a suitable node.
6. The kubelet on each selected node asks the container runtime to obtain the image and start the container.
7. Networking components give the Pod network connectivity.
8. The kubelet and controllers report status back to the API server.

Several components cooperate asynchronously. The API server can accept the Deployment even when no node currently has enough capacity. Acceptance and successful execution are separate facts.

What “self-healing” means

Kubernetes is often described as self-healing. In practical terms, this means it can perform actions such as:

- Restarting a container according to the Pod’s restart policy.
- Replacing a failed Pod that belongs to a controller.
- Maintaining the requested number of replicas.
- Removing an unready Pod from normal Service traffic when readiness information is configured.

Self-healing restores declared structure; it does not repair application logic or data correctness. If an image contains a bug that makes every new process fail, Kubernetes may repeatedly start the same faulty image. If a database contains corrupted data, replacing its Pod does not reconstruct correct data.

What Kubernetes does not provide by itself

Kubernetes is a platform of building blocks, not a complete application lifecycle solution. By itself, it does not:

- Compile source code or build container images.
- Decide an organization’s testing and release process.
- Supply a complete metrics, logging, or tracing platform.
- Create backups merely because a volume is persistent.
- Guarantee zero downtime without suitable replicas, health signals, capacity, and application behavior.

- Make insecure images or excessive permissions safe.
- Determine whether every application should run on Kubernetes.

For a small single-process service on one machine, a simpler supervisor or managed application platform may be more appropriate. Kubernetes earns its complexity when its scheduling, control, portability, policy, and automation benefits solve real operational needs.

Shared responsibility

The exact boundary depends on whether a cluster is self-managed or provided as a managed service, but the following division is a useful starting point:

Participant	Typical responsibility
Application team	Application behavior, image contents, configuration needs, graceful shutdown, data semantics
Platform team	Reusable deployment patterns, policy, observability integration, cluster add-ons, developer access
Cluster administrator	Cluster lifecycle, node capacity, control-plane health where applicable, upgrades, core networking and storage integration
Cloud or infrastructure provider	The contracted infrastructure or managed-control-plane functions
Kubernetes controllers	Reconciling declared objects within the capabilities and permissions available to them

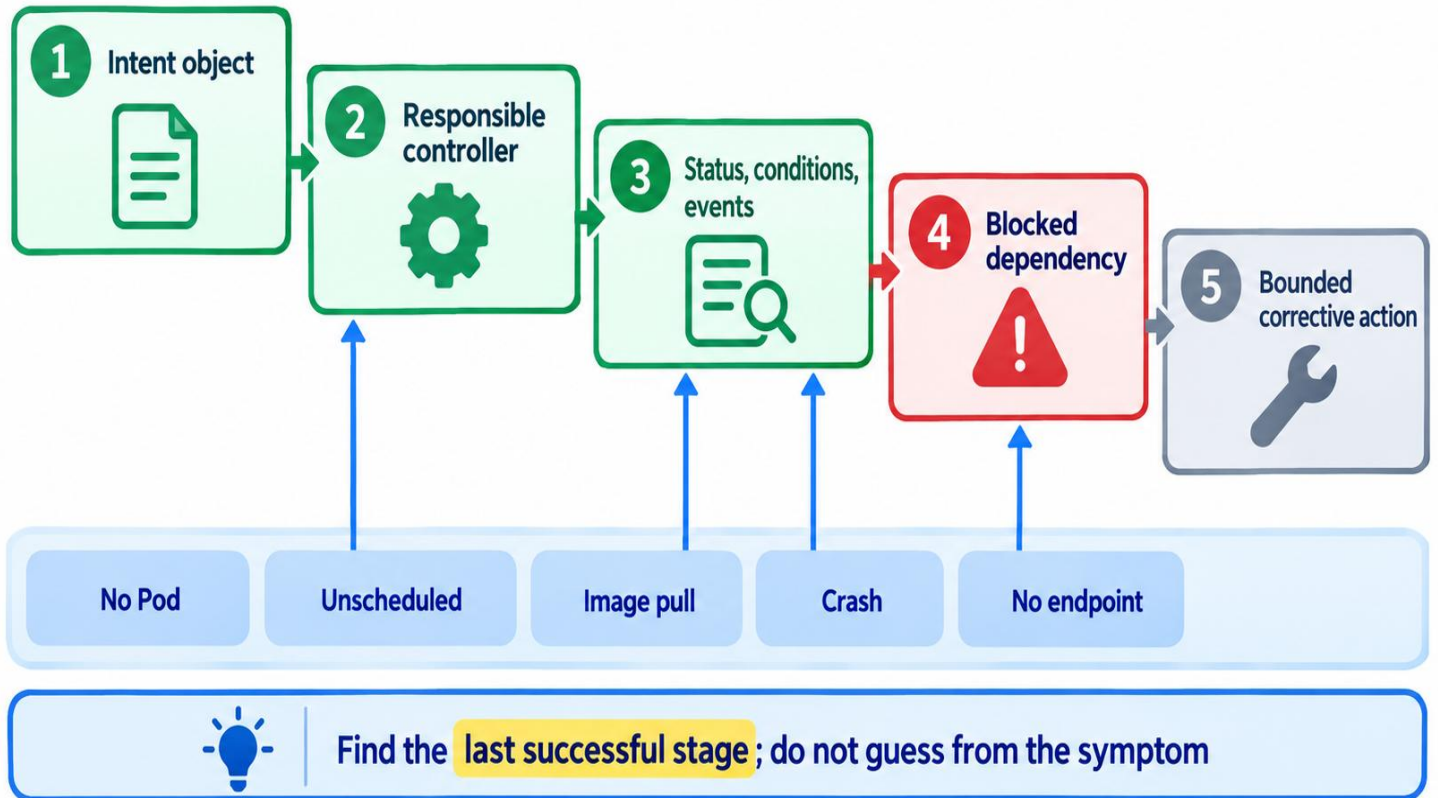
A managed cluster changes responsibilities; it does not eliminate them. The application team still owns application correctness, and the organization still owns access policy, workload configuration, data protection, and many operational decisions.

The first practical mental model

When an application is not working, avoid asking only, “Which command fixes it?” Instead ask:

Chapter 1: Kubernetes in One Picture

1-8 A dependency-first troubleshooting mental model



004

Figure 1-4: A dependency-first troubleshooting mental model

1. Which object expresses the desired state?
2. Which controller is responsible for reconciling it?
3. What actual state is reported in status, conditions, and events?
4. Which dependency prevents the next step?

For example, a Deployment with zero available Pods may be blocked at different stages: no Pod was created, a Pod was not scheduled, the image could not be pulled, the process crashed, or traffic routing did not select the Pod. Each stage has different evidence and a different owner.

The durable skill is tracing the control path, not memorizing a symptom-to-command list.

Chapter Summary

Kubernetes is an API-driven collection of control loops that coordinates declared workloads across cluster resources; it restores declared structure but does not repair application logic or eliminate operational responsibility.

Review Questions

1. How do desired state and actual state differ, and what closes the gap between them?
2. Why does an accepted API request not prove that an application is running or healthy?
3. Which responsibilities remain with application, platform, and infrastructure teams?

Chapter 2 — Container, Linux, and Network Foundations

What You'll Be Able to Do

After this chapter, you will be able to:

- Distinguish an image, registry, container, and operating-system process.
- Explain the Linux, resource, network, storage, and YAML concepts that Kubernetes builds upon.
- Identify process inputs, outputs, listening ports, and data that must outlive a container.

Kubernetes orchestrates processes

A **Kubernetes workload ultimately becomes one or more operating-system processes**. Containers package and isolate those processes, but they still consume CPU and memory, open network sockets, receive signals, write files, and return exit codes.

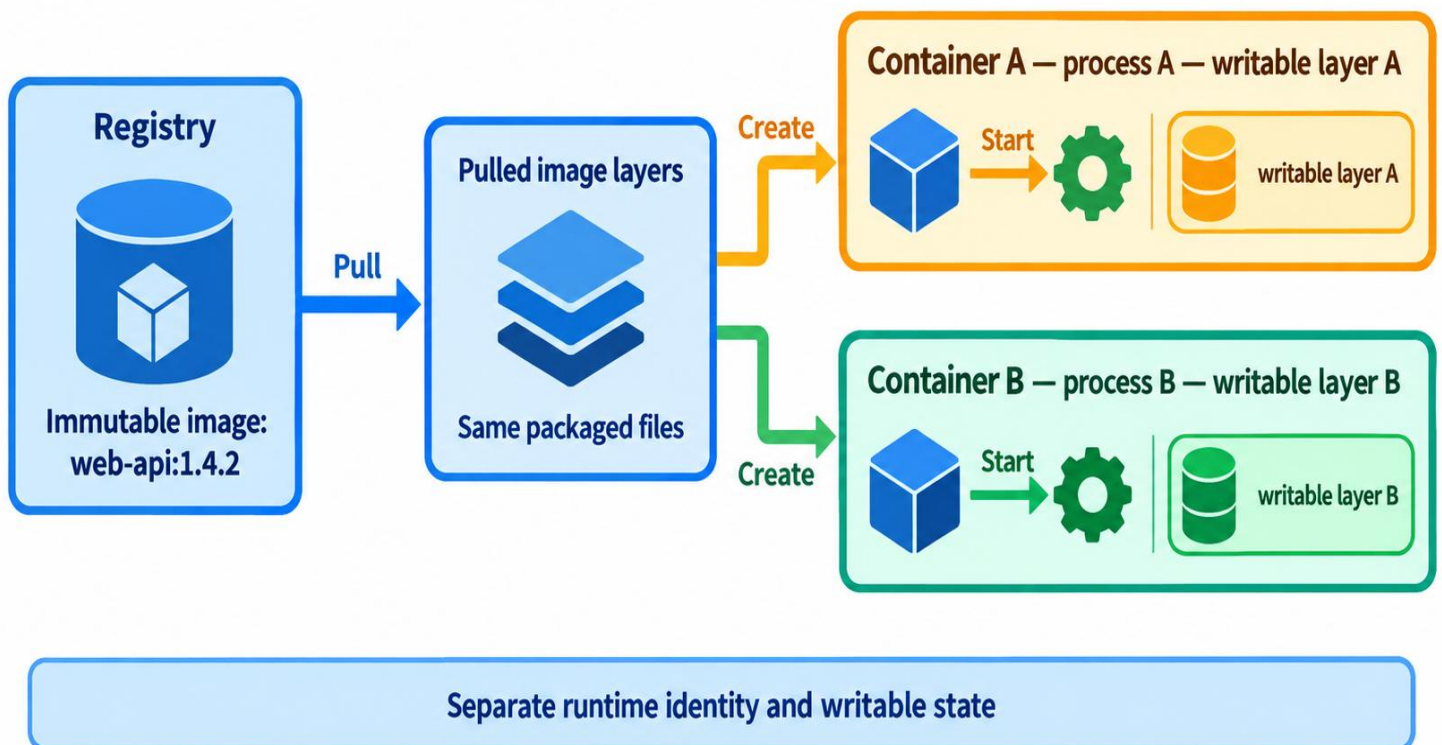
Understanding these foundations makes Kubernetes behavior much less mysterious.

Image, registry, container, and process

A **container image** is an immutable package of filesystem layers and metadata used to create containers. A **registry** is a service that stores and distributes container images. A **container** is a running or stopped instance created from an image. A **process** is an executing program managed by the operating-system kernel.

Chapter 2: Container, Linux, and Network Foundations

2-2 Image, registry, container, and process lifecycle



005

Figure 2-1: Image, registry, container, and process lifecycle

A simple analogy is a recipe and a prepared meal:

- The image is the recorded recipe plus packaged ingredients.
- The registry is the repository where recipes are stored.
- A container is one prepared instance.
- The running process is the active work being performed inside that instance.

Many containers can be created from the same image. They begin with the same packaged files, but their writable state and runtime identity are separate.

An image reference commonly has these parts:

```
registry.example.com/team/web-api:1.4.2
```

- `registry.example.com` identifies the registry.
- `team/web-api` identifies the repository.
- `1.4.2` is a tag, which is a human-readable reference that may be movable unless the registry enforces immutability.

An image can also be selected by a **digest**, a content-derived identifier such as `sha256:...`. A digest is useful when an exact, immutable image is required.

A container is not a small virtual machine

A virtual machine normally includes a complete guest operating-system kernel. Linux containers usually share the host kernel while receiving isolated views of processes, networking, mounts, users, and other resources.

Chapter 2: Container, Linux, and Network Foundations

2-3 Container isolation versus a virtual machine

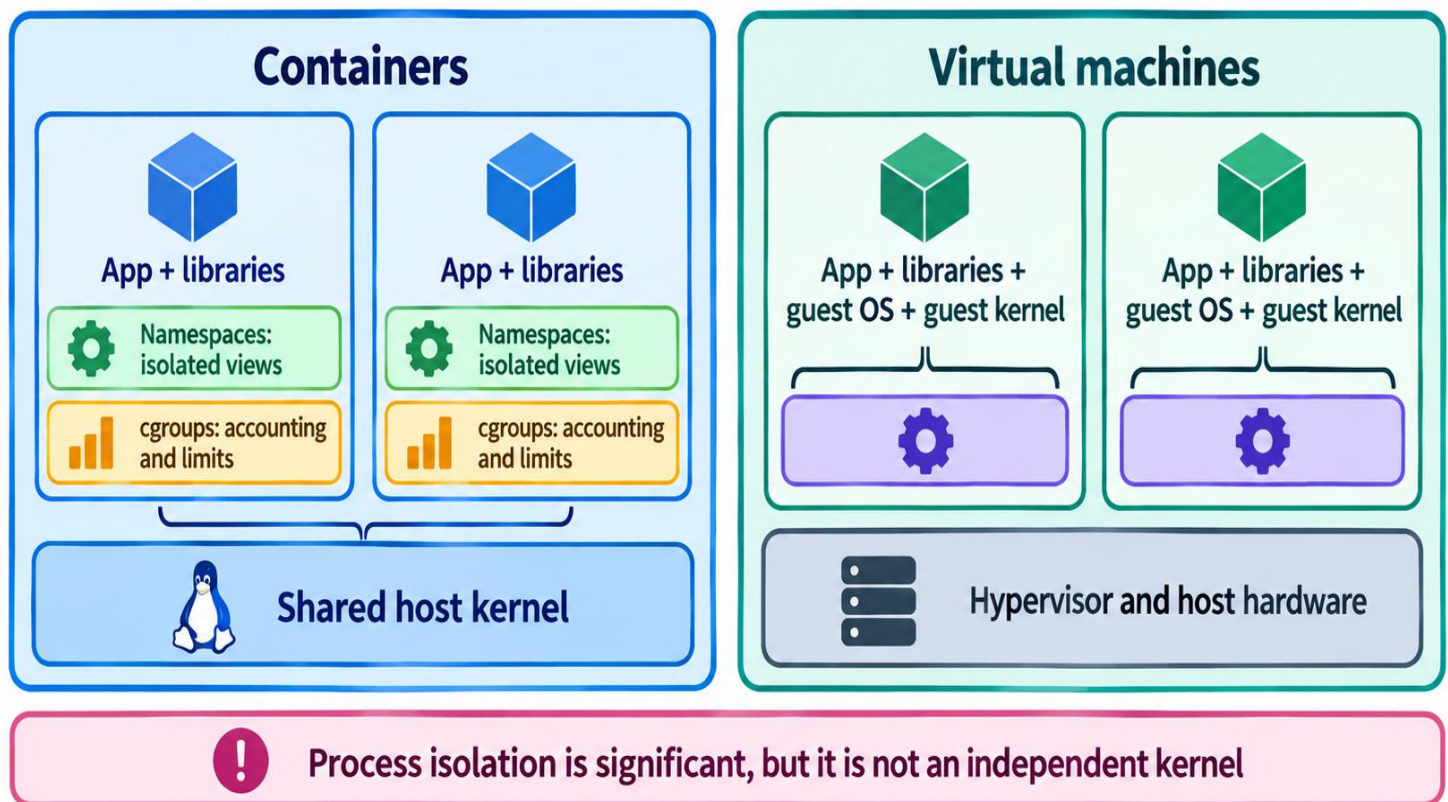


Figure 2-2: Container isolation versus a virtual machine

Two kernel mechanisms are central at a high level:

- A **namespace** gives a process an isolated view of a resource, such as process IDs or network interfaces.
- A **control group**, usually shortened to **cgroup**, accounts for and controls resource use such as CPU and memory.

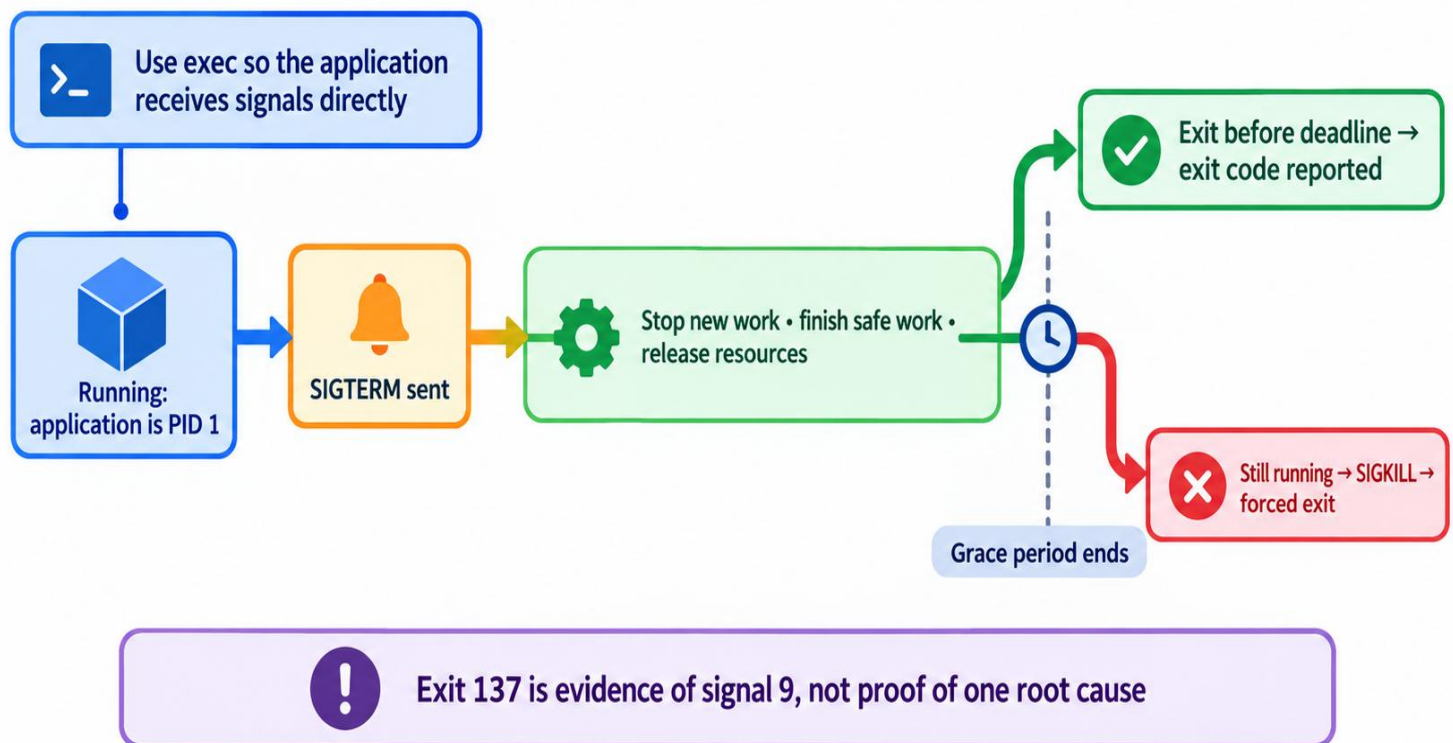
The isolation is significant, but it is not identical to a hardware virtual machine boundary. Containers are processes with controlled views and limits, not independent kernels.

PID 1, signals, and exit codes

Inside a container's process namespace, the main process commonly has process ID 1, known as **PID 1**. This process has special responsibilities, including handling termination signals correctly and reaping completed child processes.

Chapter 2: Container, Linux, and Network Foundations

2-4 PID 1 termination and exit sequence



007

Figure 2-3: PID 1 termination and exit sequence

When Kubernetes stops a normal Linux container, the expected sequence is broadly:

1. The container's main process receives a termination signal, normally `SIGTERM`.
2. The application should stop accepting new work, finish or hand off current work where appropriate, and release resources.
3. If it has not exited before the grace period ends, the runtime forcefully stops it, normally with `SIGKILL`.

An **exit code** is the integer a process returns when it finishes. By convention, `0` represents success and nonzero values represent some form of failure, although the precise meaning is application-specific. An exit caused by a signal may be presented using a shell convention such as `128 + signal number`; for example, `137` often indicates a process was killed with signal 9. Treat that as evidence to investigate, not as proof of one specific cause.

A common container mistake is using a wrapper script that does not forward signals to the real application. A robust shell wrapper usually replaces itself with the application by using `exec`:

```
#!/bin/sh
```

```
exec /app/web-server
```

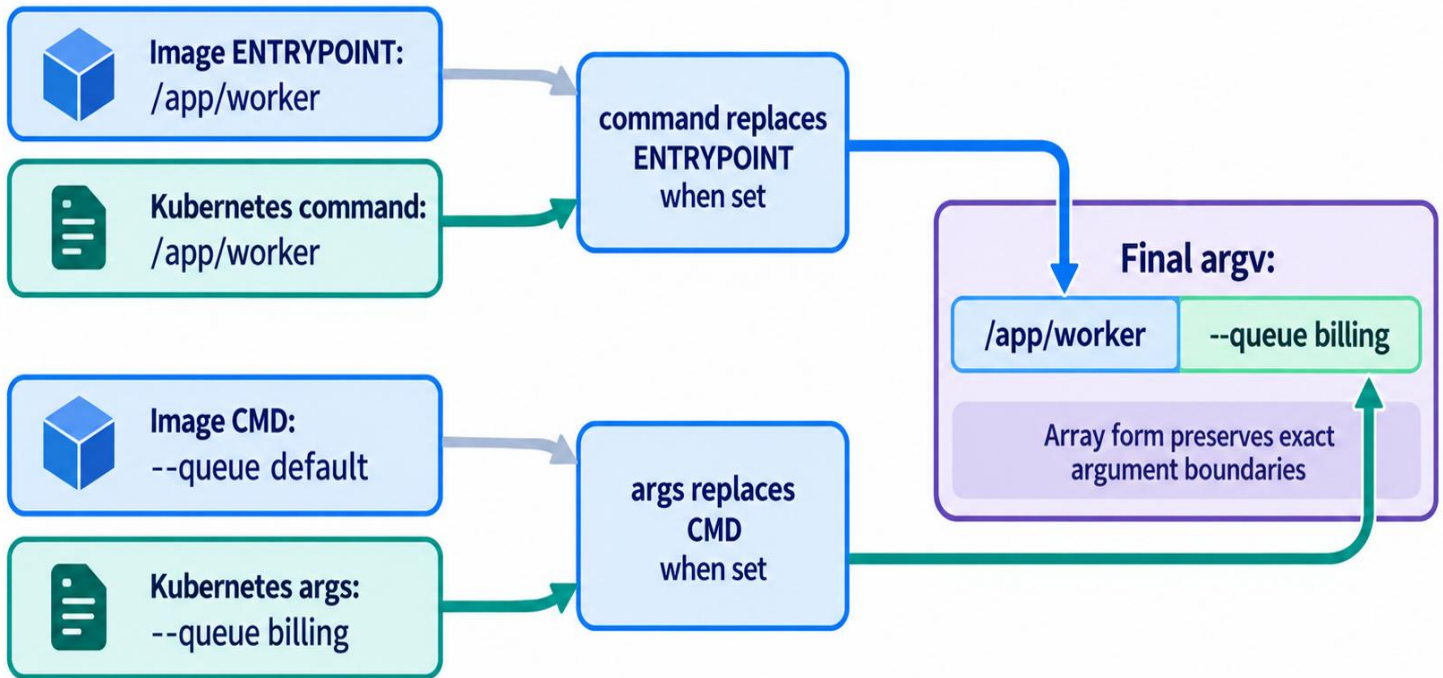
The `exec` line makes the application become the main process, allowing it to receive signals directly.

Command and arguments

An image can define a default executable and default arguments. Kubernetes can override them with a container's `command` and `args` fields.

Chapter 2: Container, Linux, and Network Foundations

2-5 How image defaults and Kubernetes overrides form the process command



008

Figure 2-4: How image defaults and Kubernetes overrides form the process command

The terminology can be confusing because Kubernetes' `command` corresponds broadly to an image's entrypoint, while `args` corresponds broadly to the image's default command arguments.

```
containers:
- name: worker
  image: example/worker:1.0
  command: ["/app/worker"]
  args: ["--queue", "billing"]
```

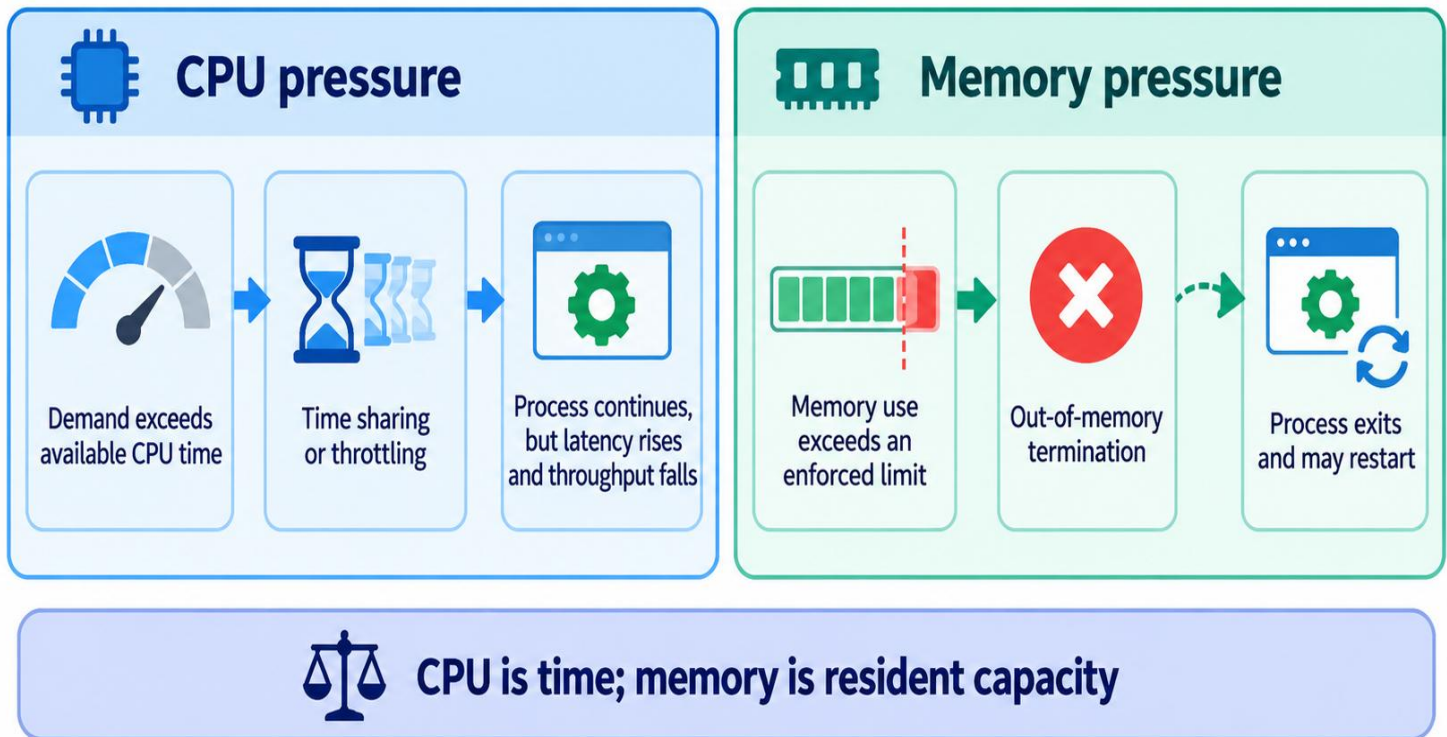
This starts `/app/worker --queue billing`. Array form is preferable when precise argument boundaries matter. It avoids accidental shell parsing, variable expansion, and quoting behavior.

CPU and memory are different kinds of resources

CPU is a time-shared resource. When demand exceeds available CPU, processes normally receive less execution time and run more slowly. Memory is capacity that must be available while data is resident. When a process exceeds an enforced memory boundary, it may be terminated rather than merely slowed.

Chapter 2: Container, Linux, and Network Foundations

2-6 CPU contention versus memory exhaustion



009

Figure 2-5: CPU contention versus memory exhaustion

This distinction becomes important when Kubernetes resource requests and limits are introduced later. For now, remember:

- CPU pressure often appears as latency or reduced throughput.
- Memory pressure can lead to process termination.
- A process may be healthy at low load and fail when its real workload increases.

Standard output and standard error

Standard output (`stdout`) and **standard error** (`stderr`) are the conventional output streams of a process. Container platforms can collect these streams without requiring the application to know the node's log-file layout.

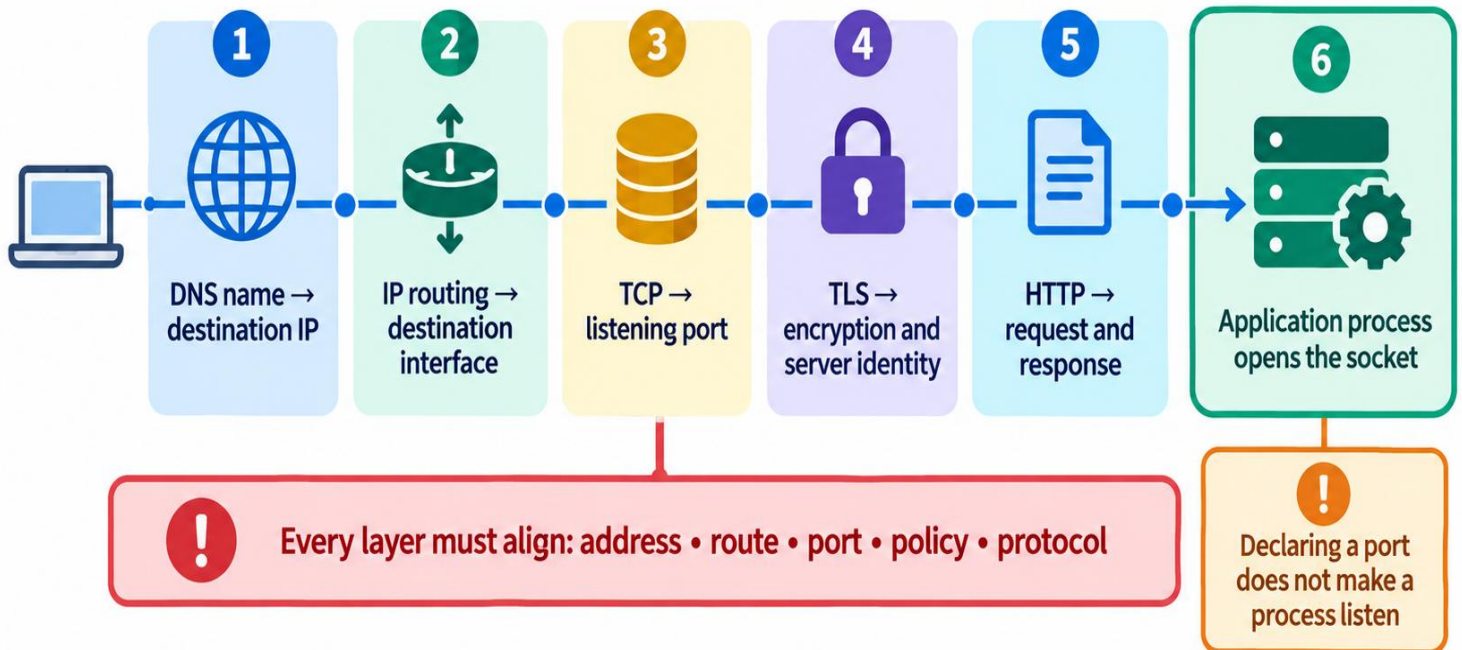
For Kubernetes-friendly applications, writing operational logs to these streams is usually simpler than writing only to a file inside the container. Cluster-level log retention still requires an external logging solution; Kubernetes does not permanently archive all container output by itself.

Networking vocabulary

An **IP address** identifies a network interface. A **port** identifies a listening endpoint on that address. **TCP** is a reliable byte-stream transport protocol. **HTTP** is an application protocol commonly carried over TCP. **TLS** protects network communication using encryption and identity verification. **DNS** translates names into network addresses or other records.

Chapter 2: Container, Linux, and Network Foundations

2-8 How IP, port, protocol, DNS, and TLS fit together



010

Figure 2-6: How IP, port, protocol, DNS, and TLS fit together

A server is reachable only when several facts align:

1. A process is running.
2. The process is listening on the expected port.
3. It is bound to an appropriate address, not only an inaccessible loopback interface.
4. Routing carries packets to the destination.
5. Firewalls or network policies allow the traffic.
6. The client uses the expected protocol.

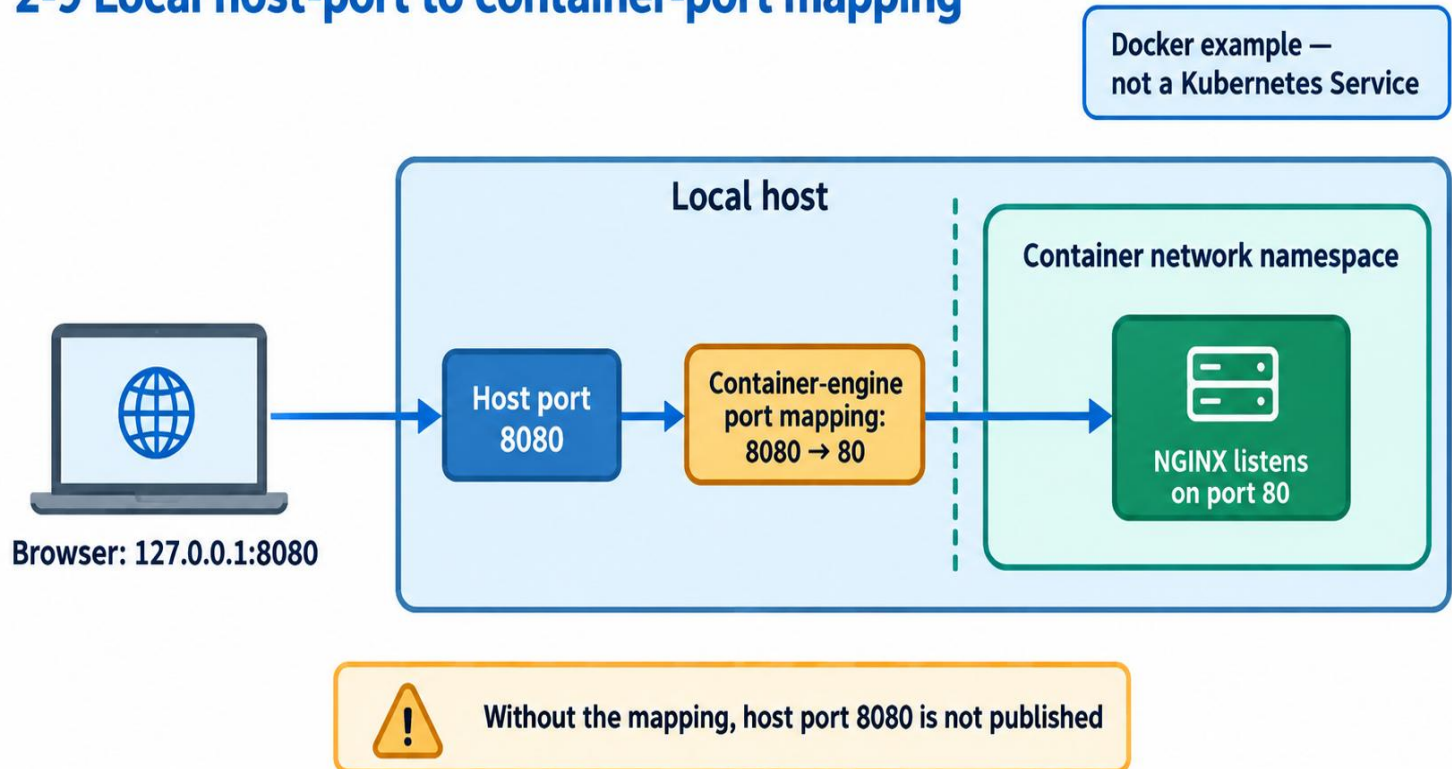
Declaring a port in a container image or Pod manifest does not cause an application to listen. It documents intent and can be referenced by other objects, but **the application process must open the socket itself**.

A local container example

The following is one Docker-based example; Podman and other compatible tools use similar concepts.

Chapter 2: Container, Linux, and Network Foundations

2-9 Local host-port to container-port mapping



011

Figure 2-7: Local host-port to container-port mapping

Prerequisites: Docker is installed, the machine can pull public images, and local port 8080 is unused.

```
docker run --name web-local --rm -p 8080:80 nginx:1.27.4-alpine
```

What it does: It creates a container from the image, maps host port 8080 to container port 80, and runs the web server in the foreground.

Expected result: A request to `http://127.0.0.1:8080/` returns the default NGINX page. The terminal displays access and error log output. Pressing `Ctrl+C` stops the container.

The `-p 8080:80` option is container-engine behavior, not a Kubernetes Service. Without that mapping, the process may still listen on port 80 inside the container's network namespace, but it is not automatically published on the host at port 8080.

To inspect the container from another terminal while it is running:

```
docker inspect web-local
```

Prerequisites: The `web-local` container is running.

What it does: It displays runtime metadata, including the image, process configuration, network settings, and mounts.

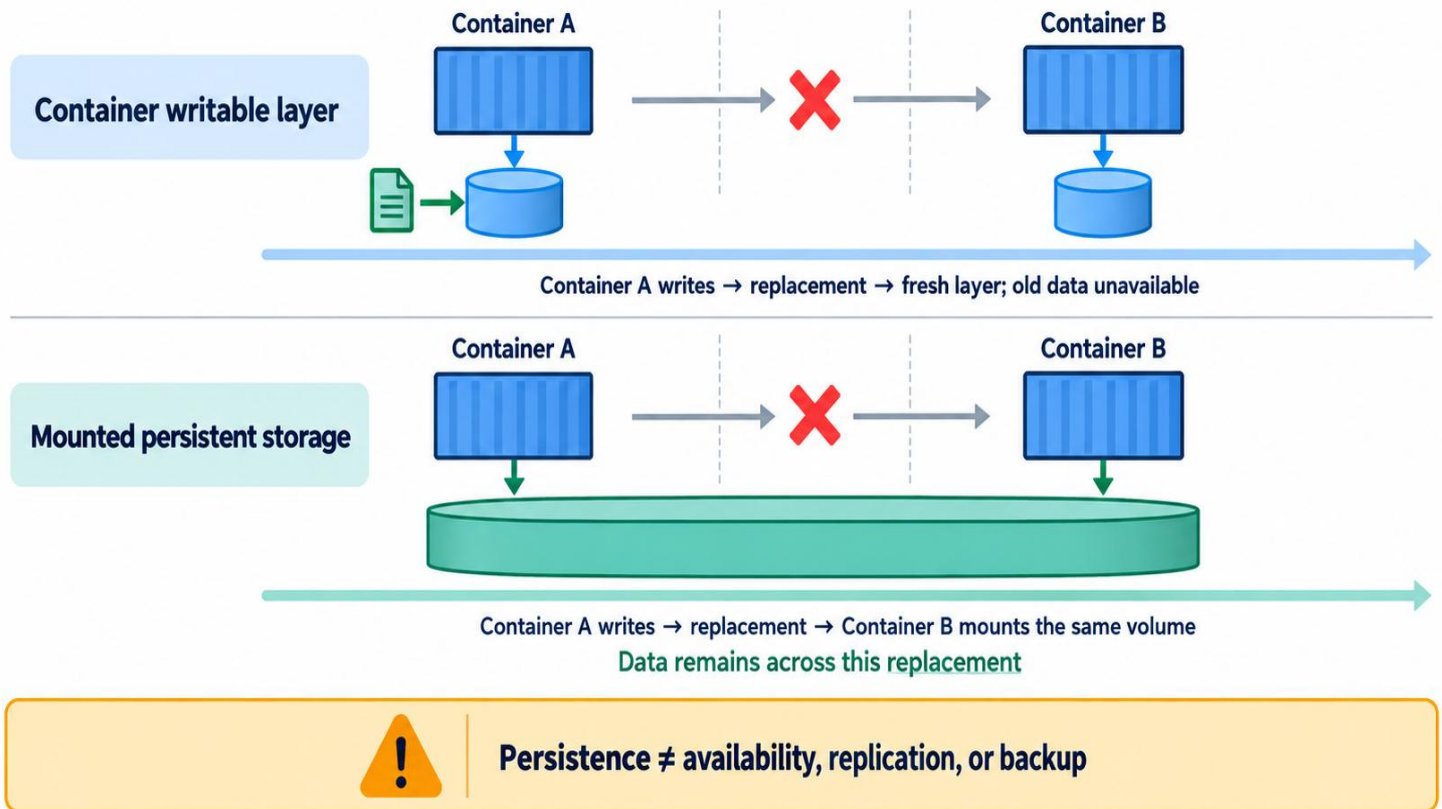
Expected result: JSON output describes one specific container instance. The output is detailed; the useful skill is locating relevant fields rather than memorizing all of them.

Ephemeral and persistent data

Ephemeral data is expected to disappear when its runtime instance is replaced. **Persistent data** must survive beyond the lifetime of a particular container or Pod.

Chapter 2: Container, Linux, and Network Foundations

2-10 Ephemeral writable layer versus persistent storage



012

Figure 2-8: Ephemeral writable layer versus persistent storage

A container usually has a writable filesystem layer. That layer is convenient for temporary files, caches, and extracted runtime content, but it belongs to that container instance. Replacing the container can produce a fresh writable layer.

A **mount** presents storage at a path in a filesystem. Mounting storage can separate data lifetime from container lifetime, but a mount is not automatically a backup.

These are different concerns:

- Persistence answers, “Does the data remain when this container or Pod is replaced?”
- Availability answers, “Can the application access the data when infrastructure fails?”
- Backup answers, “Can an earlier correct copy be restored after deletion, corruption, or disaster?”

A persistent disk may survive a Pod replacement and still have no usable backup.

Environment variables

An **environment variable** is a key-value setting inherited by a process when it starts. It is commonly used for small configuration values such as a log level or a service endpoint.

Environment variables are fixed for the lifetime of a process. Changing the external source of a variable does not rewrite the already running process environment. The process must be restarted to receive a different value unless it uses another dynamic configuration mechanism.

Environment variables are also visible through several process and debugging interfaces. They should not be treated as a universally secret channel merely because they are not printed in normal logs.

YAML essentials

Kubernetes manifests are usually written in **YAML**, a human-readable data-serialization format. The important structures are mappings, lists, and scalar values.

A mapping associates keys with values:

```
metadata:
  name: web-demo
  namespace: learning
```

A list contains ordered items introduced with `-`:

```
ports:
  - name: http
    containerPort: 80
```

Indentation defines structure. Use spaces consistently; tabs are not valid indentation in YAML.

The following fragment means that `containers` is a list with one mapping item:

```
spec:
  containers:
    - name: web
      image: nginx:1.27.4-alpine
      ports:
        - name: http
          containerPort: 80
```

YAML values have types. These values are not always equivalent:

```
enabled: true
literalText: "true"
replicas: 3
replicaText: "3"
```

`true` is a Boolean, `"true"` is a string, `3` is a number, and `"3"` is a string. Kubernetes validates fields according to the API schema, so a visually similar value of the wrong type can be rejected.

Strings that resemble numbers, Booleans, dates, or special YAML values are safer when quoted if the receiving field expects text. For environment variables, use strings:

```
env:
  - name: FEATURE_ENABLED
    value: "true"
```

Foundation check

Before moving on, the following statements should be clear:

- **An image is a package; a container is an instance; the useful work is performed by processes.**
- Container isolation uses the host kernel and is not the same as a full virtual machine.
- The main process must handle signals and exit cleanly.
- Declaring a port does not make an application listen or publish it externally.
- A container's writable layer is not durable application storage.
- YAML indentation and value types affect the object sent to the Kubernetes API.

These principles explain many Kubernetes symptoms that otherwise look platform-specific.

Chapter Summary

Kubernetes ultimately schedules and supervises Linux processes packaged as containers, so process lifecycle, resource behavior, networking, filesystems, and structured configuration remain essential foundations.

Review Questions

1. Why is a container not the same as a small virtual machine?
2. What happens when PID 1 does not handle termination signals correctly?
3. Why does writing a file inside a container not automatically make the data persistent?